1. Ganesh reddy

192110023

**19th march programs**

Easy

1. #include <iostream>

using namespace std;

// Base class Shape

class Shape {

public:

virtual void draw() const = 0; // Pure virtual function

};

// Derived class Circle

class Circle : public Shape {

private:

double radius;

public:

Circle(double r) : radius(r) {}

void draw() const override {

cout << "Drawing a circle with radius " << radius << endl;

// Add drawing code for circle here

}

};

// Derived class Rectangle

class Rectangle : public Shape {

private:

double length;

double width;

public:

Rectangle(double l, double w) : length(l), width(w) {}

void draw() const override {

cout << "Drawing a rectangle with length " << length << " and width " << width << endl;

// Add drawing code for rectangle here

}

};

int main() {

// Creating objects of Circle and Rectangle

Circle circle(5.0);

Rectangle rectangle(4.0, 6.0);

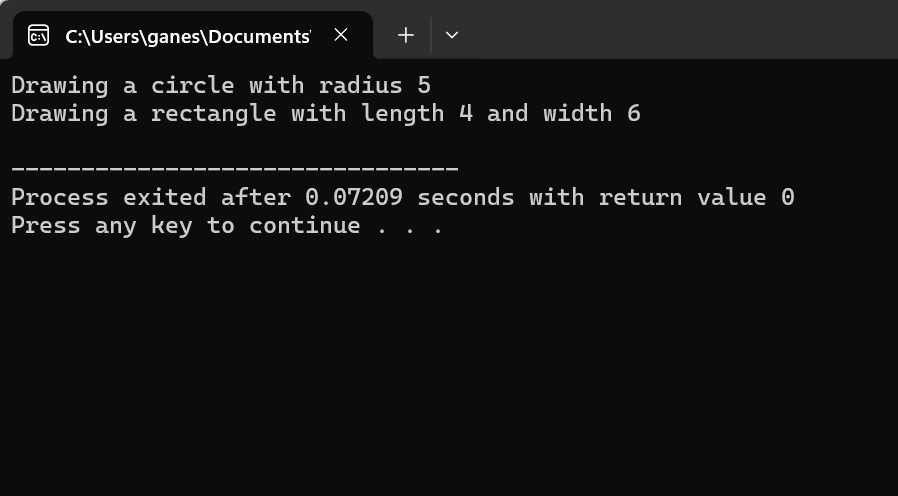
// Calling draw() for Circle and Rectangle objects

circle.draw();

rectangle.draw();

return 0;

}



2.

#include <iostream>

using namespace std;

// Abstract base class Animal

class Animal {

public:

virtual void makeSound() const = 0; // Pure virtual function

};

// Derived class Dog

class Dog : public Animal {

public:

void makeSound() const override {

cout << "Woof! Woof!" << endl;

}

};

// Derived class Cat

class Cat : public Animal {

public:

void makeSound() const override {

cout << "Meow! Meow!" << endl;

}

};

int main() {

// Creating objects of Dog and Cat

Dog dog;

Cat cat;

// Calling makeSound() for Dog and Cat objects

cout << "Dog says: ";

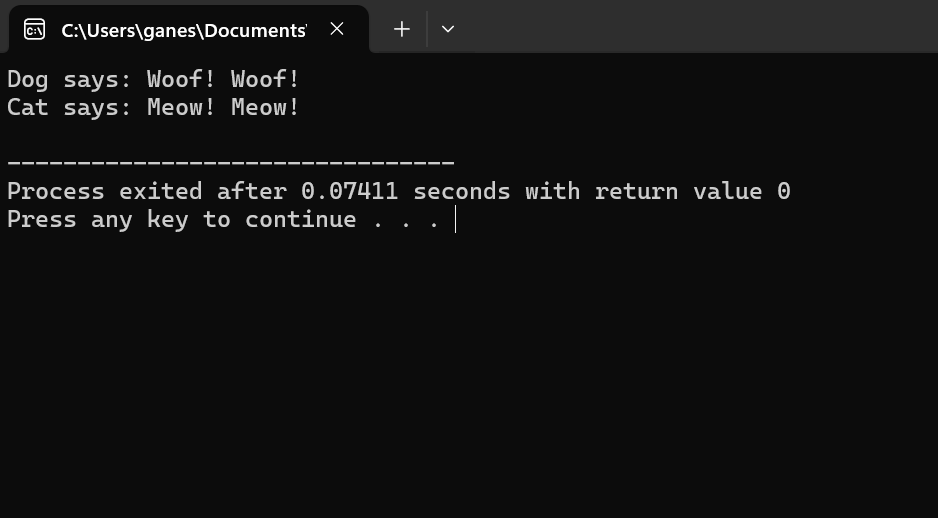
dog.makeSound();

cout << "Cat says: ";

cat.makeSound();

return 0;

}



3.

#include <iostream>

#include <string>

using namespace std;

// Base class Employee

class Employee {

protected:

string name;

int employeeID;

double salary;

public:

Employee(const string& n, int id, double sal) : name(n), employeeID(id), salary(sal) {}

// Virtual function to display employee information

virtual void displayInfo() const {

cout << "Employee ID: " << employeeID << endl;

cout << "Name: " << name << endl;

cout << "Salary: $" << salary << endl;

}

};

// Derived class Manager

class Manager : public Employee {

private:

string department;

public:

Manager(const string& n, int id, double sal, const string& dept) : Employee(n, id, sal), department(dept) {}

// Override displayInfo() to include manager-specific details

void displayInfo() const override {

Employee::displayInfo(); // Call base class displayInfo()

cout << "Department: " << department << endl;

}

};

// Derived class Clerk

class Clerk : public Employee {

private:

string supervisor;

public:

Clerk(const string& n, int id, double sal, const string& sup) : Employee(n, id, sal), supervisor(sup) {}

// Override displayInfo() to include clerk-specific details

void displayInfo() const override {

Employee::displayInfo(); // Call base class displayInfo()

cout << "Supervisor: " << supervisor << endl;

}

};

int main() {

// Creating objects of Manager and Clerk

Manager manager("John Doe", 1001, 60000.0, "Sales");

Clerk clerk("Jane Smith", 2001, 35000.0, "Manager: John Doe");

// Displaying information of Manager and Clerk

cout << "Manager Information:" << endl;

manager.displayInfo();

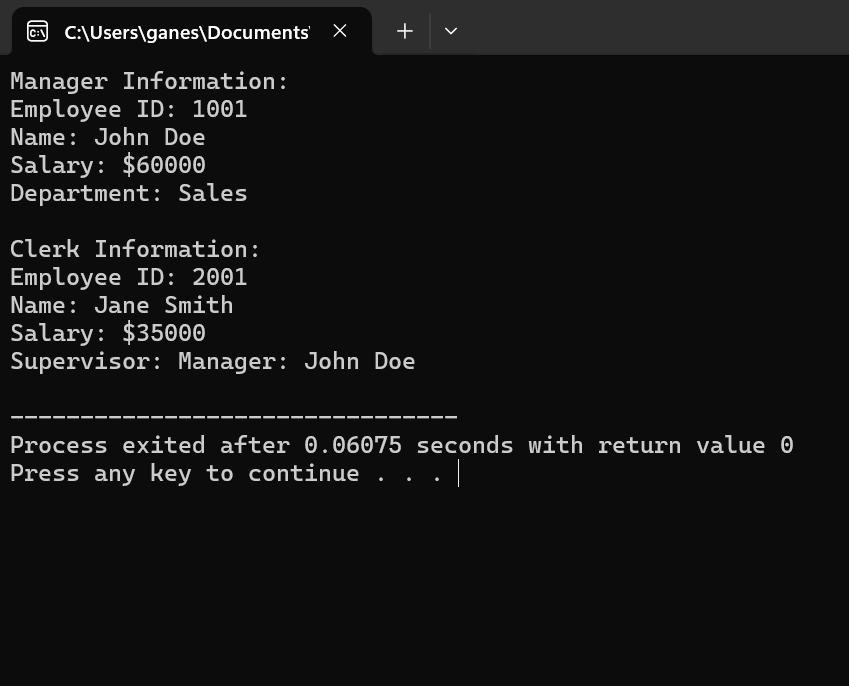
cout << endl;

cout << "Clerk Information:" << endl;

clerk.displayInfo();

return 0;

}



4.

#include <iostream>

using namespace std;

// Abstract base class BankAccount

class BankAccount {

protected:

double balance;

public:

BankAccount(double initialBalance) : balance(initialBalance) {}

// Pure virtual function for withdrawal

virtual void withdraw(double amount) = 0;

};

// Derived class SavingsAccount

class SavingsAccount : public BankAccount {

private:

double interestRate;

public:

SavingsAccount(double initialBalance, double rate) : BankAccount(initialBalance), interestRate(rate) {}

// Override withdraw() function for savings account

void withdraw(double amount) override {

if (amount <= balance) {

balance -= amount;

cout << "Withdrawal of $" << amount << " from Savings Account. Remaining balance: $" << balance << endl;

} else {

cout << "Insufficient funds in Savings Account." << endl;

}

}

};

// Derived class CurrentAccount

class CurrentAccount : public BankAccount {

private:

double overdraftLimit;

public:

CurrentAccount(double initialBalance, double limit) : BankAccount(initialBalance), overdraftLimit(limit) {}

// Override withdraw() function for current account

void withdraw(double amount) override {

if (amount <= balance + overdraftLimit) {

balance -= amount;

cout << "Withdrawal of $" << amount << " from Current Account. Remaining balance: $" << balance << endl;

} else {

cout << "Exceeds overdraft limit. Transaction not allowed." << endl;

}

}

};

int main() {

// Creating objects of SavingsAccount and CurrentAccount

SavingsAccount savings(1000.0, 0.05);

CurrentAccount current(500.0, 200.0);

// Performing withdrawals

savings.withdraw(200.0);

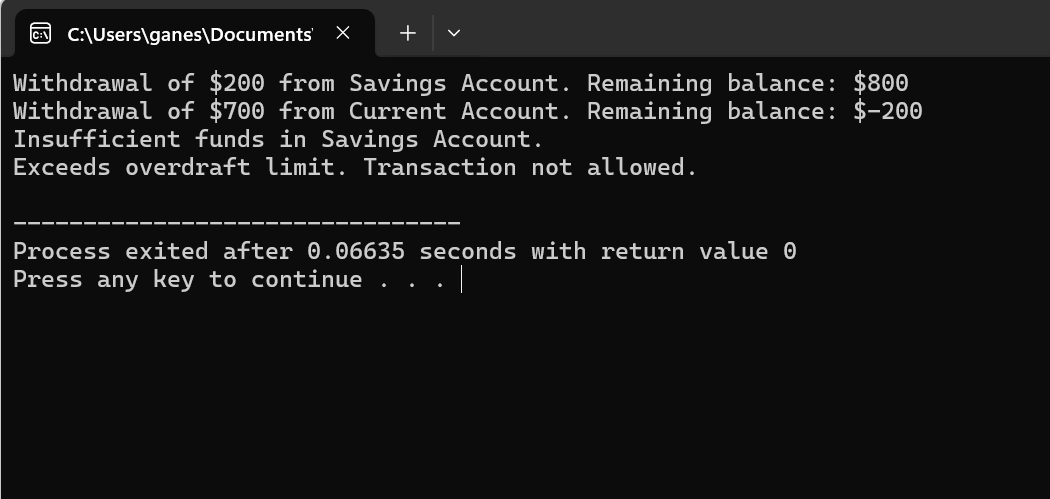
current.withdraw(700.0);

savings.withdraw(1200.0); // Attempting to withdraw more than balance

current.withdraw(800.0);

return 0;

}



5.

#include <iostream>

#include <stdexcept>

using namespace std;

int main() {

try {

int numerator, denominator;

cout << "Enter the numerator: ";

cin >> numerator;

cout << "Enter the denominator: ";

cin >> denominator;

if (denominator == 0) {

throw runtime\_error("Division by zero is not allowed.");

}

double result = static\_cast<double>(numerator) / denominator;

cout << "Result of division: " << result << endl;

} catch (const runtime\_error& e) {

cerr << "Error: " << e.what() << endl;

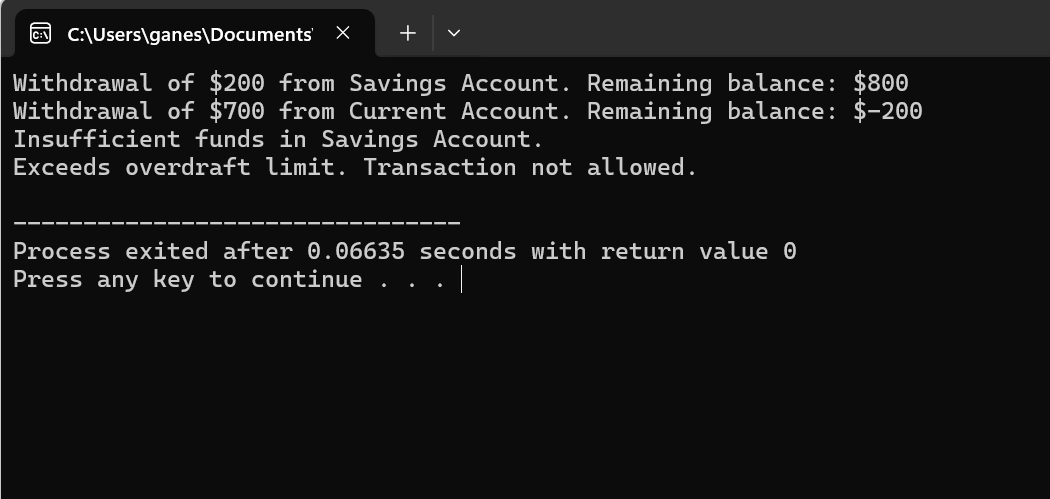
} catch (...) {

cerr << "An unexpected error occurred." << endl;

}

return 0;

}



6.

#include <iostream>

#include <stdexcept>

using namespace std;

int calculateAverage(int array[], int size) {

if (size == 0) {

throw invalid\_argument("Array size is zero. Cannot calculate average.");

}

int sum = 0;

for (int i = 0; i < size; ++i) {

sum += array[i];

}

return sum / size;

}

int main() {

int array1[] = {1, 2, 3, 4, 5};

int size1 = sizeof(array1) / sizeof(array1[0]);

try {

int average1 = calculateAverage(array1, size1);

cout << "Average of array1: " << average1 << endl;

} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

}

int array2[] = {};

int size2 = sizeof(array2) / sizeof(array2[0]);

try {

int average2 = calculateAverage(array2, size2);

cout << "Average of array2: " << average2 << endl;

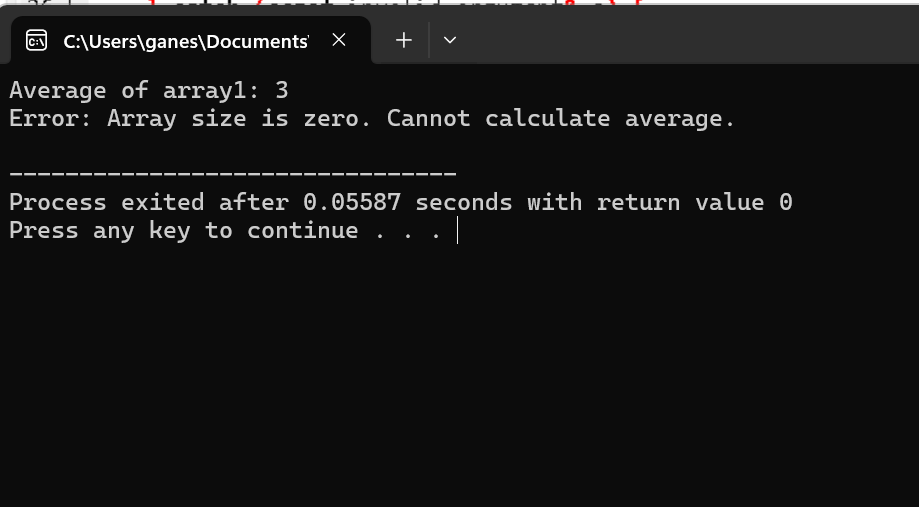
} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

}

return 0;

}



7.

#include <iostream>

#include <stdexcept>

using namespace std;

int main() {

try {

int age;

cout << "Please enter your age: ";

cin >> age;

if (age < 0 || age > 150) {

throw out\_of\_range("Invalid age entered. Age must be between 0 and 150.");

}

cout << "Your age is: " << age << endl;

} catch (const out\_of\_range& e) {

cerr << "Error: " << e.what() << endl;

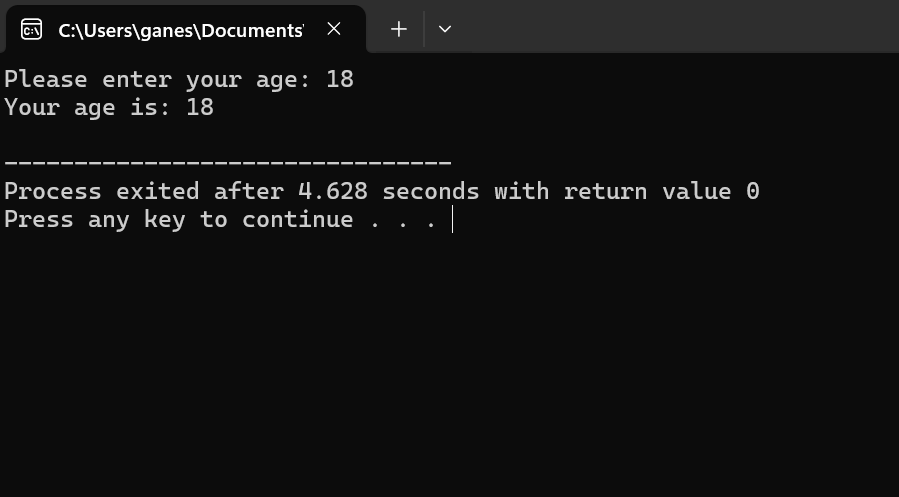
} catch (...) {

cerr << "An unexpected error occurred." << endl;

}

return 0;

}



8.

#include <iostream>

#include <stdexcept>

using namespace std;

int main() {

try {

int num;

cout << "Please enter an integer between 1 and 100: ";

cin >> num;

if (num < 1 || num > 100) {

throw out\_of\_range("Integer is not in the range [1, 100]");

}

cout << "The entered integer is: " << num << endl;

} catch (const out\_of\_range& e) {

cerr << "Error: " << e.what() << endl;

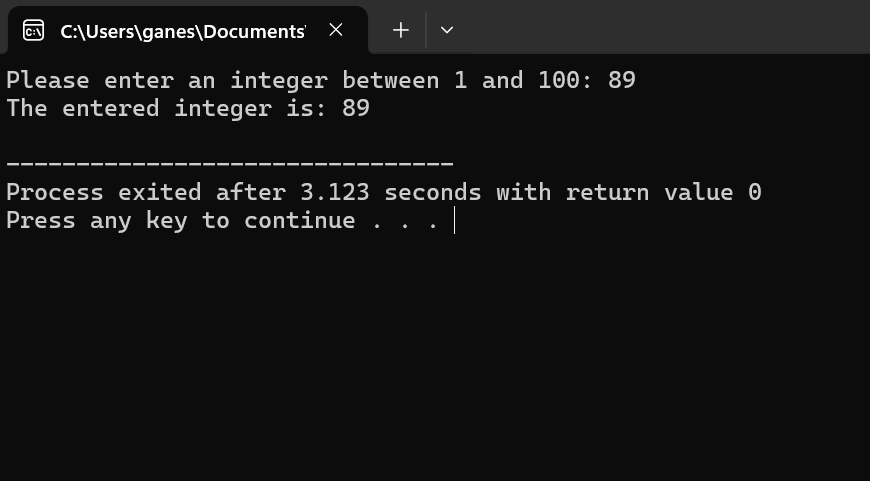
} catch (...) {

cerr << "An unexpected error occurred." << endl;

}

return 0;

}



9.

#include <iostream>

#include <cmath> // For sqrt function

#include <stdexcept> // For exception handling

using namespace std;

int main() {

try {

int num;

cout << "Please enter an integer: ";

cin >> num;

if (num < 0) {

throw invalid\_argument("Cannot calculate square root of a negative number.");

}

double squareRoot = sqrt(num);

cout << "Square root of " << num << " is: " << squareRoot << endl;

} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

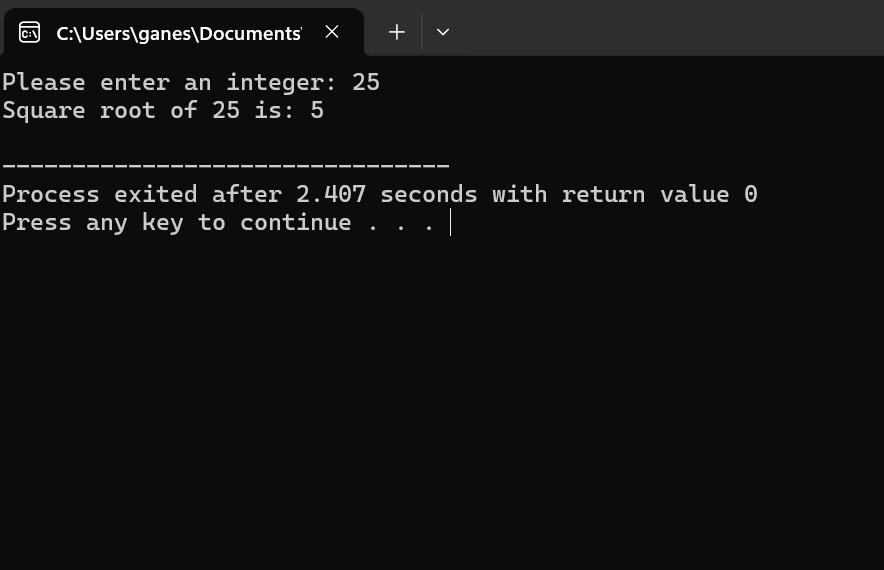
} catch (...) {

cerr << "An unexpected error occurred." << endl;

}

return 0;

}



10.

#include <iostream>

#include <vector>

#include <stdexcept>

using namespace std;

int findElement(const vector<int>& vec, int target) {

if (vec.empty()) {

throw runtime\_error("Vector is empty. Cannot search for element.");

}

for (size\_t i = 0; i < vec.size(); ++i) {

if (vec[i] == target) {

return i; // Return the index of the target element

}

}

return -1; // Default value indicating target element not found

}

int main() {

vector<int> vec = {10, 20, 30, 40, 50};

try {

int index = findElement(vec, 30);

if (index != -1) {

cout << "Element found at index: " << index << endl;

} else {

cout << "Element not found." << endl;

}

} catch (const runtime\_error& e) {

cerr << "Error: " << e.what() << endl;

}

vector<int> emptyVec;

try {

int index = findElement(emptyVec, 30);

if (index != -1) {

cout << "Element found at index: " << index << endl;

} else {

cout << "Element not found." << endl;

}

} catch (const runtime\_error& e) {

cerr << "Error: " << e.what() << endl;

}

return 0;

}

**Medium programs**

11.

#include <iostream>

#include <stdexcept>

using namespace std;

// Function to calculate factorial using recursion

unsigned long long factorial(int n) {

if (n < 0) {

throw invalid\_argument("Cannot calculate factorial of a negative number.");

}

return (n == 0 || n == 1) ? 1 : n \* factorial(n - 1);

}

int main() {

// Test case to calculate factorial of a positive number

int num = 5;

try {

unsigned long long result = factorial(num);

cout << "Factorial of " << num << " is: " << result << endl;

// Validation: Calculate factorial iteratively to compare the result

unsigned long long expected = 1;

for (int i = 1; i <= num; ++i) {

expected \*= i;

}

if (result == expected) {

cout << "Validation: Result is correct." << endl;

} else {

cout << "Validation: Result is incorrect." << endl;

}

} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

}

// Test case for a negative number

num = -5;

try {

unsigned long long result = factorial(num);

cout << "Factorial of " << num << " is: " << result << endl;

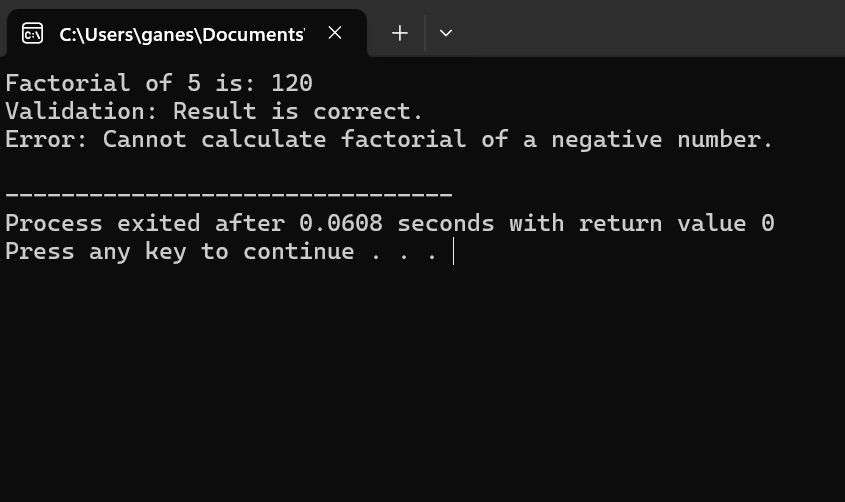
} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

}

return 0;

}



12.

#include <iostream>

#include <stdexcept>

using namespace std;

class TemperatureConverter {

public:

// Member function to convert Celsius to Fahrenheit

double convertToFahrenheit(double celsius) {

if (celsius < -273.15) {

throw invalid\_argument("Celsius temperature cannot be below absolute zero (-273.15°C).");

}

return (celsius \* 9.0 / 5.0) + 32.0;

}

};

int main() {

TemperatureConverter converter;

// Test case: Convert a valid Celsius temperature to Fahrenheit

double celsius = 25.0;

try {

double fahrenheit = converter.convertToFahrenheit(celsius);

cout << "Celsius: " << celsius << "°C -> Fahrenheit: " << fahrenheit << "°F" << endl;

} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

}

// Test case: Convert a Celsius temperature below absolute zero

celsius = -300.0;

try {

double fahrenheit = converter.convertToFahrenheit(celsius);

cout << "Celsius: " << celsius << "°C -> Fahrenheit: " << fahrenheit << "°F" << endl;

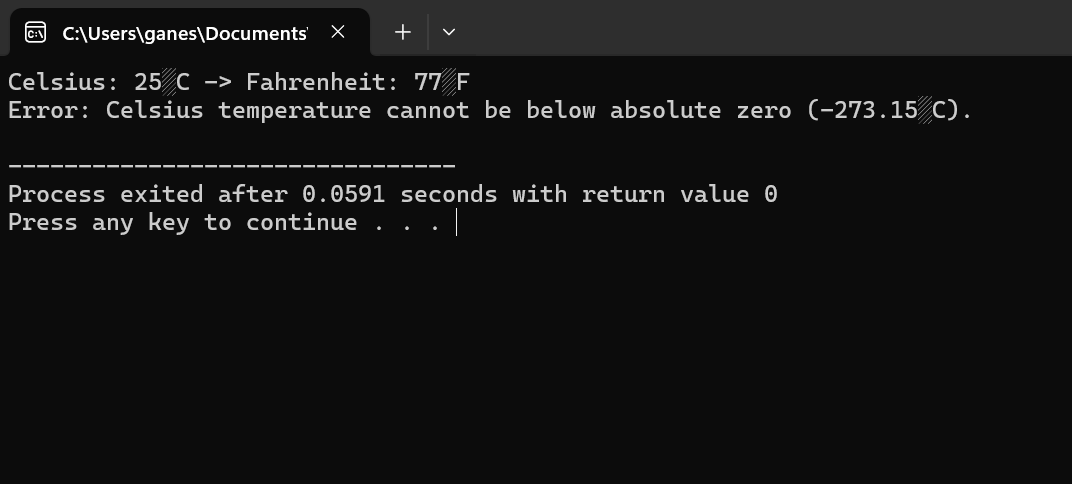
} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

}

return 0;

}



13.

#include <iostream>

#include <stdexcept>

using namespace std;

class Array {

private:

int \*data;

int size;

int capacity;

public:

// Constructor

Array(int initialCapacity) : size(0), capacity(initialCapacity) {

if (initialCapacity <= 0) {

throw invalid\_argument("Capacity must be a positive integer.");

}

data = new int[capacity];

}

// Destructor

~Array() {

delete[] data;

}

// Function to insert element into the array

void insert(int element) {

if (size == capacity) {

// Increase capacity by doubling it

capacity \*= 2;

int \*temp = new int[capacity];

for (int i = 0; i < size; ++i) {

temp[i] = data[i];

}

delete[] data;

data = temp;

}

data[size++] = element;

}

// Function to retrieve element by index

int get(int index) const {

if (index < 0 || index >= size) {

throw out\_of\_range("Invalid index.");

}

return data[index];

}

};

int main() {

try {

// Create an array with initial capacity 5

Array arr(5);

// Insert elements into the array

arr.insert(10);

arr.insert(20);

arr.insert(30);

arr.insert(40);

arr.insert(50);

// Retrieve elements by index and print them

cout << "Elements in the array:" << endl;

for (int i = 0; i < 5; ++i) {

cout << "Index " << i << ": " << arr.get(i) << endl;

}

// Attempt to access an invalid index

cout << "Attempting to access an invalid index:" << endl;

cout << "Index 10: " << arr.get(10) << endl; // This should throw an exception

} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

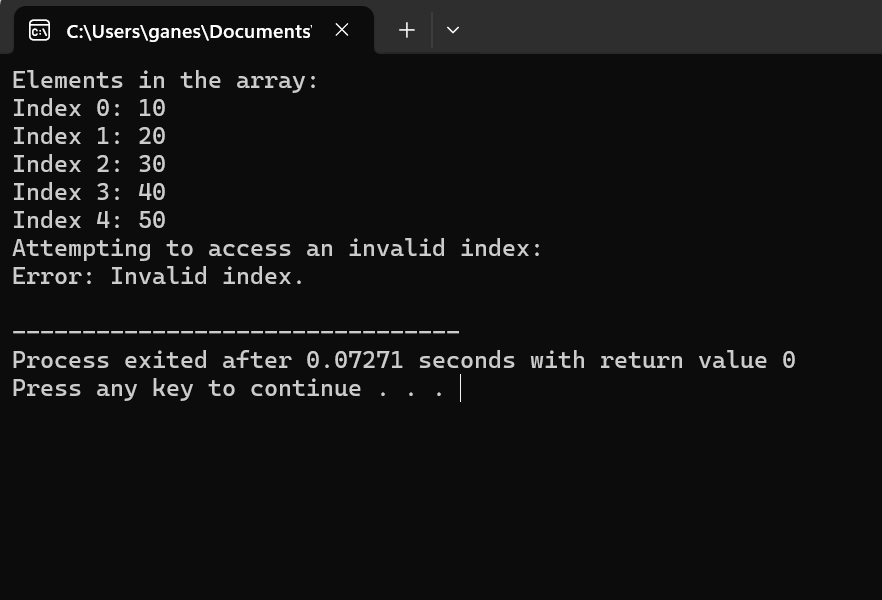
} catch (const out\_of\_range& e) {

cerr << "Error: " << e.what() << endl;

}

return 0;

}



14.

#include <iostream>

#include <stdexcept>

using namespace std;

class Student {

private:

int rollNumber;

int marks;

public:

// Constructor

Student(int roll, int m) : rollNumber(roll), marks(m) {}

// Getters

int getRollNumber() const {

return rollNumber;

}

int getMarks() const {

return marks;

}

};

int main() {

try {

int numStudents;

cout << "Enter the number of students: ";

cin >> numStudents;

if (numStudents <= 0) {

throw invalid\_argument("Number of students must be a positive integer.");

}

// Create a dynamic array of Student objects

int \*students = new Student[numStudents];

// Populate the array with student details

for (int i = 0; i < numStudents; ++i) {

int roll, marks;

cout << "Enter roll number for student " << i + 1 << ": ";

cin >> roll;

cout << "Enter marks for student " << i + 1 << ": ";

cin >> marks;

students[i] = Student(roll, marks);

}

// Display student details

cout << "Student details:" << endl;

for (int i = 0; i < numStudents; ++i) {

cout << "Student " << i + 1 << " - Roll Number: " << students[i].getRollNumber()

<< ", Marks: " << students[i].getMarks() << endl;

}

// Free the allocated memory

delete[] students;

} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

} catch (const bad\_alloc& e) {

cerr << "Memory allocation failed: " << e.what() << endl;

}

return 0;

}

15.

#include <iostream>

#include <vector>

#include <stdexcept>

using namespace std;

class Matrix {

private:

vector<vector<int>> data;

int rows;

int cols;

public:

// Constructor

Matrix(int numRows, int numCols, const vector<vector<int>>& initialData) : rows(numRows), cols(numCols) {

if (numRows <= 0 || numCols <= 0) {

throw invalid\_argument("Number of rows and columns must be positive integers.");

}

if (numRows != initialData.size() || numCols != initialData[0].size()) {

throw invalid\_argument("Invalid initial data dimensions.");

}

data = initialData;

}

// Function to perform matrix addition

Matrix add(const Matrix& other) const {

if (rows != other.rows || cols != other.cols) {

throw invalid\_argument("Matrices have incompatible dimensions for addition.");

}

vector<vector<int>> resultData(rows, vector<int>(cols, 0));

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < cols; ++j) {

resultData[i][j] = data[i][j] + other.data[i][j];

}

}

return Matrix(rows, cols, resultData);

}

// Function to perform matrix multiplication

Matrix multiply(const Matrix& other) const {

if (cols != other.rows) {

throw invalid\_argument("Matrices have incompatible dimensions for multiplication.");

}

vector<vector<int>> resultData(rows, vector<int>(other.cols, 0));

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < other.cols; ++j) {

for (int k = 0; k < cols; ++k) {

resultData[i][j] += data[i][k] \* other.data[k][j];

}

}

}

return Matrix(rows, other.cols, resultData);

}

// Function to display the matrix

void display() const {

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < cols; ++j) {

cout << data[i][j] << " ";

}

cout << endl;

}

}

};

int main() {

try {

// Test case for matrix addition

vector<vector<int>> data1 = {{1, 2}, {3, 4}};

Matrix matrix1(2, 2, data1);

vector<vector<int>> data2 = {{5, 6}, {7, 8}};

Matrix matrix2(2, 2, data2);

cout << "Matrix 1:" << endl;

matrix1.display();

cout << "Matrix 2:" << endl;

matrix2.display();

cout << "Matrix Addition:" << endl;

Matrix resultAddition = matrix1.add(matrix2);

resultAddition.display();

// Test case for matrix multiplication

vector<vector<int>> data3 = {{1, 2, 3}, {4, 5, 6}};

Matrix matrix3(2, 3, data3);

vector<vector<int>> data4 = {{7, 8}, {9, 10}, {11, 12}};

Matrix matrix4(3, 2, data4);

cout << "Matrix 3:" << endl;

matrix3.display();

cout << "Matrix 4:" << endl;

matrix4.display();

cout << "Matrix Multiplication:" << endl;

Matrix resultMultiplication = matrix3.multiply(matrix4);

resultMultiplication.display();

} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

}

return 0;

}

16.

#include <iostream>

#include <fstream>

#include <stdexcept>

using namespace std;

void displayFileContents(const string& filename) {

ifstream file(filename);

if (!file.is\_open()) {

throw runtime\_error("Unable to open file: " + filename);

}

string line;

cout << "File Contents:" << endl;

while (getline(file, line)) {

cout << line << endl;

}

file.close();

}

int main() {

try {

// Test case to read and display the contents of a valid file

string filename = "example.txt";

displayFileContents(filename);

} catch (const runtime\_error& e) {

cerr << "Error: " << e.what() << endl;

}

return 0;

}

**Hard programs**

**17.**

#include <iostream>

#include <vector>

using namespace std;

// Base class Shape

class Shape {

public:

// Pure virtual function draw

virtual void draw() const = 0;

};

// Derived class Circle from Shape

class Circle : public Shape {

public:

// Implementation of draw function for Circle

void draw() const override {

cout << "Drawing Circle" << endl;

}

};

// Derived class Rectangle from Shape

class Rectangle : public Shape {

public:

// Implementation of draw function for Rectangle

void draw() const override {

cout << "Drawing Rectangle" << endl;

}

};

// Derived class Triangle from Shape

class Triangle : public Shape {

public:

// Implementation of draw function for Triangle

void draw() const override {

cout << "Drawing Triangle" << endl;

}

};

// Function to dynamically create an array of pointers to Shape objects

// Populate it with objects of different shapes, and call the draw() function for each object

void drawShapes(const vector<Shape\*>& shapes) {

for (const auto& shape : shapes) {

shape->draw();

}

}

int main() {

// Create an array of pointers to Shape objects

vector<Shape\*> shapes;

// Populate it with objects of different shapes

Circle circle;

Rectangle rectangle;

Triangle triangle;

// Add objects to the vector

shapes.push\_back(&circle);

shapes.push\_back(&rectangle);

shapes.push\_back(&triangle);

// Call draw() function for each object

drawShapes(shapes);

// Release memory

for (const auto& shape : shapes) {

delete shape;

}

return 0;

}

18.

#include <iostream>

using namespace std;

// Base class Shape

class Shape {

public:

// Virtual function displayArea

virtual void displayArea() const = 0;

};

// Derived class Rectangle from Shape

class Rectangle : public Shape {

private:

double length;

double width;

public:

// Constructor

Rectangle(double l, double w) : length(l), width(w) {}

// Implementation of displayArea function for Rectangle

void displayArea() const override {

cout << "Area of Rectangle: " << length \* width << endl;

}

};

// Derived class Circle from Shape

class Circle : public Shape {

private:

double radius;

public:

// Constructor

Circle(double r) : radius(r) {}

// Implementation of displayArea function for Circle

void displayArea() const override {

cout << "Area of Circle: " << 3.14159 \* radius \* radius << endl;

}

};

int main() {

// Create objects of derived classes

Rectangle rectangle(5.0, 3.0);

Circle circle(2.5);

// Create pointers to the base class

Shape\* ptrRectangle = &rectangle;

Shape\* ptrCircle = &circle;

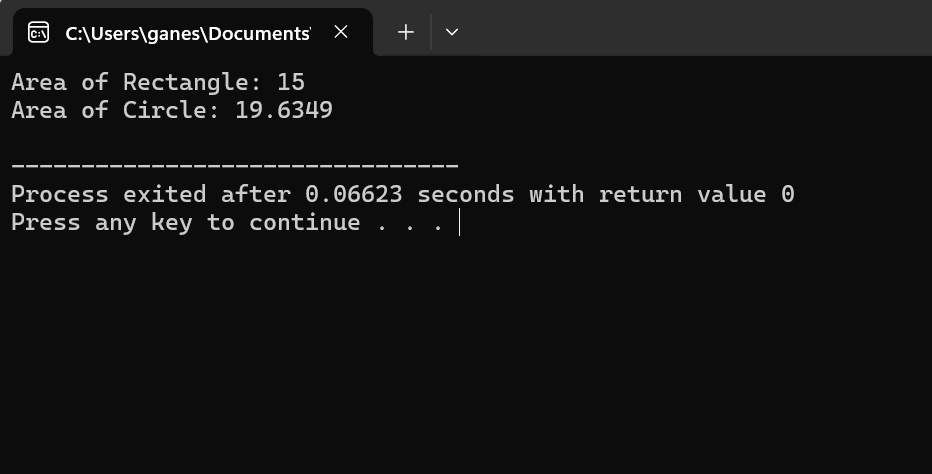
// Call displayArea() function using pointers to the base class

ptrRectangle->displayArea();

ptrCircle->displayArea();

return 0;

}



19.

#include <iostream>

#include <fstream>

#include <stdexcept>

#include <vector>

#include <limits>

using namespace std;

// Function to read integers from a file and calculate their sum

int sumIntegersFromFile(const string& filename) {

ifstream file(filename);

if (!file.is\_open()) {

throw runtime\_error("Unable to open file: " + filename);

}

int sum = 0;

int num;

while (file >> num) {

// Check for integer overflow

if ((num > 0 && sum > numeric\_limits<int>::max() - num) || (num < 0 && sum < numeric\_limits<int>::min() - num)) {

file.close();

throw overflow\_error("Arithmetic overflow detected while calculating sum.");

}

sum += num;

}

// Check for invalid input format

if (!file.eof()) {

file.close();

throw invalid\_argument("Invalid input format in file: " + filename);

}

file.close();

return sum;

}

int main() {

try {

// Test case with a valid input file

string validFile = "valid\_input.txt";

int sumValid = sumIntegersFromFile(validFile);

cout << "Sum of integers in '" << validFile << "': " << sumValid << endl;

// Test case with an invalid input file

string invalidFile = "invalid\_input.txt";

int sumInvalid = sumIntegersFromFile(invalidFile);

cout << "Sum of integers in '" << invalidFile << "': " << sumInvalid << endl;

} catch (const runtime\_error& e) {

cerr << "Error: " << e.what() << endl;

} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

} catch (const overflow\_error& e) {

cerr << "Error: " << e.what() << endl;

}

return 0;

}

20.

#include <iostream>

#include <stdexcept>

using namespace std;

class BankAccount {

private:

int accountNumber;

double balance;

public:

// Constructor

BankAccount(int accNumber, double initialBalance) : accountNumber(accNumber), balance(initialBalance) {}

// Member function to perform a withdrawal operation

void withdraw(double amount) {

if (amount <= 0) {

throw invalid\_argument("Invalid withdrawal amount. Amount must be greater than zero.");

}

if (amount > balance) {

throw runtime\_error("Insufficient balance for withdrawal.");

}

// Additional checks for specific error scenarios can be added here

// Withdraw the amount

balance -= amount;

cout << "Withdrawal successful. New balance: " << balance << endl;

}

};

int main() {

// Create a bank account with initial balance

BankAccount account(123456, 1000.0);

try {

// Test the withdraw() function with various withdrawal amounts

account.withdraw(500); // Successful withdrawal

account.withdraw(100); // Successful withdrawal

account.withdraw(800); // Attempt to withdraw more than the balance

account.withdraw(0); // Attempt to withdraw zero amount

account.withdraw(-100); // Attempt to withdraw negative amount

} catch (const invalid\_argument& e) {

cerr << "Invalid argument: " << e.what() << endl;

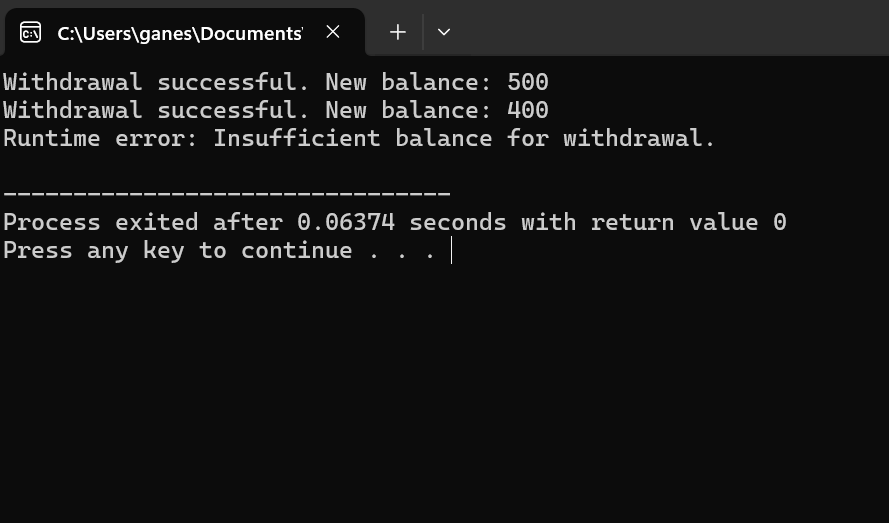
} catch (const runtime\_error& e) {

cerr << "Runtime error: " << e.what() << endl;

}

return 0;

}

21.

#include <iostream>

#include <limits>

#include <stdexcept>

using namespace std;

int main() {

const int SENTINEL = -1;

cout << "Enter integers (enter " << SENTINEL << " to end):" << endl;

try {

while (true) {

int num;

cin >> num;

// Check if the input is the sentinel value

if (num == SENTINEL) {

break;

}

// Check for invalid input type

if (cin.fail()) {

cin.clear(); // Clear error flags

cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // Discard invalid input

throw invalid\_argument("Invalid input type. Please enter an integer.");

}

// Check for out-of-range values

if (num < 0 || num > 100) {

throw out\_of\_range("Input value is out of range. Please enter a value between 0 and 100.");

}

// Process the valid input

cout << "Input: " << num << endl;

}

} catch (const invalid\_argument& e) {

cerr << "Error: " << e.what() << endl;

} catch (const out\_of\_range& e) {

cerr << "Error: " << e.what() << endl;

}

return 0;

}

